Advanced Programming Techniques
ExAdvPT
Winter Term 2018/2019

Assignment sheet B

Assignments that are marked with *StudOn submission* are **mandatory** and must be submitted via StudOn in time – please see the StudOn page for deadlines.

Fun with matrices

Your successful implementation of last assignment's matrix class has attracted some attention of students from other departments. In detail, some guys from the math division seem to have a use for it. They claim, that they have a solver for Partial Differential Equations (PDEs) that is just missing a well-written matrix class. (... and a vector class, but that is merely a generalization of the matrix class.) In their current project, they solve a 1D finite differences discretization of Poisson's equation

$$\Delta u = b,$$

$$b = sin(2\pi x).$$

Fortunately, you don't need to go into the math too deep as there is already a setup routine and a solver implementation. Your task is finishing what they started.

1 Coding: Setting up the solver

Start by downloading the solver implementation (Solver.cpp) which will serve as test case for your implementations. As you can see, the initialization of the utilized matrix and vectors is already present and doesn't have not be adapted. However, it assumes that the employed data type can easily be changed, i.e. is modeled as a template parameter. The solver itself is already fully functional, that is if suitable `Matrix` and `Vector` classes are provided.

mandatory Extend your matrix class such that it holds the data type of its elements as template parameter.

mandatory Set up a `Vector` class. This should be quite similar to a $N \times 1$ matrix, but at this point copy-pasting the matrix class is probably easier than inheriting it.

mandatory Check that you vector and matrix classes provide all required functionalities. This will most likely require the implementation of three additional functions:

- `Matrix::inverseDiagonal` which calculates the inverse of the diagonal of itself. In other words: extract the diagonal of the matrix and invert the resulting diagonal matrix. It is sufficient to implement this function for square matrices (remember suitable `assert`'s). Please also refrain from implementing a general matrix inversion!

- Multiplication of a matrix and a vector. The function signature should look like this:

```
Vector<T> operator* (const Vector<T> & o) const;
```

- `Vector::l2Norm`, which calculates the L2 norm of itself. An initial implementation could look like this:

```
double l2Norm ( ) const {
  double norm = 0.;
  for (int i = 0; i < size_; ++i)
    norm += data_[i] * data_[i];
  return sqrt(norm);
}
```

  optional Think about possible enhancements and use `std::accumulate` to realize a much more concise implementation.

mandatory Set up a suitable Makefile and think about suitable compile flags. The latter should include `-O3`.

2 Checking results

StudOn submission

To roughly verify the results of your implementation run the solver for varying problem sizes (17, 33, 65, 129). Afterward, submit the required number of iterations for each test case via StudOn. **Don't continue implementing unless you have done this!**

3 Coding: Checking performance

Your colleagues are happy with your implementation, but somehow you are not satisfied. The main reason is that you suspect untapped performance possibilities. Have a look at `std::chrono` and use its functionality to time the solver in order to assess its performance.

4 Coding: Improving performance

As you expect, you find out that the solver is quite slow and, even worse, scales horribly. After a quick analysis, you identify the matrix operations as a possible cause. Moreover, the matrix in the sample problem has a very special structure[1] but this is not exploited at all! *Verify this by printing the matrix at hand.* As a possible counter measure, you devise a way of improving performance by introducing a stencil class. However, you don't want to break the general applicability of the solver which is why you decide to add a common interface in the form of an abstract base class.

mandatory Download the initial interface implementation (`MatrixLike.h`) from StudOn and extend your `Matrix` class to implement the interface (i.e. derive from `MatrixLike`). Also change the solve function to accept a reference to a `MatrixLike` instance:

```
void solve (const MatrixLike<T, MatrixImpl>& A,
            const Vector<T>& b, Vector<T>& u)
```

[1] `en.wikipedia.org/wiki/Tridiagonal_matrix`

If you measure your solver again, you will most likely observe a severe performance penalty. Think about possible reasons and adapt the `MatrixLike` interface accordingly. (hint: there are two ways of doing this and one of them is simply removing the `()` operator from the interface.)

After fixing the interface, it is time to implement the `Stencil` class.

mandatory Download the skeleton (`Stencil.h`) from StudOn and implement the prepared functions. Implement the `testStencil` function in `Solver.cpp` and verify that the results of the solver (i.e. printed residuals and iteration counts) do not change when using the `Stencil` class instead of the `Matrix` class.

optional Make use of `std::find_if` when implementing `Stencil::inverseDiagonal` to identify the entry with zero offset.

5 Coding: Checking performance ... again

After successfully implementing the stencil class it is time to compare execution times. Think about possible reasons for observed differences.

6 Coding: Improving usability

As a last step, you decide to extend the `Vector` class for improved usability.

optional Implement a constructor for the `Vector` class taking a `std::function` used to initialize its contents. This eliminates the old initialization routine

```
Vector<double> b(numGridPoints, 0.);
for (int x = 0; x < numGridPoints; ++x) {
  b(x) = sin(2. * PI * (x / (double)(numGridPoints - 1)));
}
```

and replaces it with a new expression based on a lambda expression, e.g.

```
Vector<double> b(numGridPoints, /*insert lambda here*/ );
```

Even more fun with matrices

Some of your colleagues claim, that they know a way to improve performance even further. It is your job to verify this claim.

7 Coding: Extending your classes

Their proposal is this: when adding more information to the template arguments, such as number of rows and columns for matrices or number of elements for vectors, everything should get more robust and (potentially) faster. This means, at the example of the `Matrix` class, annotating target dimensions in this fashion:

```
template<typename T, size_t rows, size_t cols>
class Matrix
  : public MatrixLike<T, Matrix<T, rows, cols>, rows, cols >
{ ... }
```

mandatory Your task is to extend your `Matrix`, `Vector` and `Stencil` classes to implement this concept. Naturally, this usually requires an adaptation of the `MatrixLike` interface and the solver function as well. After you implemented your extension, the signature of the solve method should be given by

```
template<typename T, typename MatrixImpl, size_t numPoints>
void solve (
  const MatrixLike<T, MatrixImpl, numPoints, numPoints>& A,
  const Vector<T, numPoints>& b,
  Vector<T, numPoints>& u) { ... }
```

optional At this point it may become feasible to merge your `Vector` class into your `Matrix` class. One option to do this is given by implementing the `Vector` class as a partial specialization of the `Matrix` class. Think about pro's and con's of this approach.
Another idea, which is probably more suitable for the task at hand, is using a type alias, e.g.

```
template<typename T, size_t rows_>
using Vector = Matrix<T, rows_, 1>;
```

This requires moving some functions, most likely the access operators and `l2norm`, to the matrix class. It is highly advisable to use (`static_`)`assert`'s to restrict their usage for actual vectors, i.e. matrices with only one column.

8 Coding: Refactoring your classes

StudOn submission

After annotating the respective matrix and vector dimensions, some minor code refactoring is in order.

mandatory Take a look at your implemented `assert`'s and think about which of them can be eliminated since their validity is secured by the combination of template parameters, and which of them can be replaced by `static_assert`'s. Furthermore, since all sizes are known at compile time, restructure your memory management such that the custom memory management is replaced. For this task, you can use, e.g., `std::array`.

mandatory Now it's time to download the provided `MatrixTest.cpp`, `SolverTest.cpp` and `SolverShort.cpp`. These files *must compile (with flags `-std=c++14 -Wall -pedantic -O3`) and run successfully* on the Linux CIP-pool computers. Run `SolverShort.cpp` with `valgrind` to test for invalid heap memory accesses. Compile and link `SolverShort.cpp` with the g++ flag `-fsanitize=address` to check for invalid stack memory accesses (a `CMakeLists.txt` is provided on studon).

After you've completed your task, add your solutions for `Matrix.h`, `Vector.h` (empty if merged into `Matrix.h`) and `Stencil.h` as well as your adaptations of `Solver.cpp` and `MatrixLike.h` to an archive named `assignmentB.zip` and upload it to StudOn.

9 Coding: Extending timing functionalities

By now you should have (at least) two test cases, one for the matrix-based solver and one for the stencil-based one. Timing both probably required copy-pasting the respective routine.

optional Improve reusability by wrapping the timing script in a function with the following signature:

```
double measureTime (std::function<void( )> toMeasure) { ... }
```

As you can see, the function takes a parameterless std::function as input and returns the measured time. However, usually parameters are required for the function to be measured. As you might already know from the last assignment, one solution for this problem is using lambda functions with capture clauses.

optional Adapt your code to make use of the newly implemented routine. Additionally, refrain from using lambdas this time and make use of std::bind instead.

Function resolution

10 Function overloading

StudOn submission

mandatory Demonstrate your basic knowledge in the resolution of functions by taking a quick StudOn Quiz ("‘Function resolution - basics"’).

11 Inherited functions

StudOn submission

mandatory Demonstrate your advanced knowledge in the resolution of functions by taking a quick StudOn Quiz ("‘Function resolution - classes and structs"’).

12 Template function specialization

StudOn submission

mandatory Demonstrate your even more advanced knowledge in the resolution of functions by taking a quick StudOn Quiz ("‘Function resolution - templates"’).

13 Mixed concepts

optional Show off your extreme knowledge in the resolution of functions by taking a quick StudOn Quiz ("‘Function resolution - challenge"’).